

Introducción

Una *shell* es un [intérprete de lenguaje de órdenes](#). *Csh* es el nombre de un intérprete de órdenes particular de la Distribución de Software UNIX la Universidad de Berkeley (BSD). El objetivo principal de *csh* es traducir las órdenes escritas en la línea de comandos por medio de un [terminal](#) en acciones de cómputo o invocaciones a otros programas. *Csh* es realmente un programa de usuario similar a cualquier otro del sistema. Con suerte... *csh* será un programa muy útil para interactuar con el sistema UNIX.

Junto a este documento, querrás consultar una copia del [Manual del programador de UNIX](#). La documentación de *csh* en el manual ofrece descripción completa de todas las características de la shell y oficia de referencia para todas las interrogantes sobre la misma. Muchas palabras de este documento aparecen en *cursiva*; son palabras importantes como nombres de órdenes, y palabras que tienen un significado especial al hablar de la shell y UNIX. Muchas de las palabras están definidas en un [glosario](#) al final de este documento. Si no sabes lo que significa una palabra, consulta el glosario.

1. Uso de C shell en la Terminal

Normalmente la shell que estarás usando cuando inicies sesión a `texto-plano.xyz` será [/bin/ksh](#). De hecho, la mayoría de lo dictado en este tutorial aplica a ella, así también como a [/bin/sh](#).

[C Shell](#) ya no es el intérprete de órdenes por defecto en `texto-plano.xyz`, pero podrás utilizarla anidada en otra shell cualquiera para seguir este tutorial histórico. Ingresa a ella primero:

```
csh
```

Normalmente, recibirás el *prompt* por defecto de *csh*: `%` . ¡Ya podrás continuar con el tutorial! Cuando termines y desees cerrar la C Shell y volver a tu intérprete anterior, asegúrate de ingresar el comando:

```
exit
```

La noción básica de comandos

Si bien tiene un conjunto de *funciones integradas* que puede realizar directamente, una shell en UNIX actúa principalmente como un medio a través del cual se invocan otros *programas*. La mayoría de las órdenes provocan la ejecución de programas que, de hecho, son externos al shell. La shell se distingue así de los intérpretes de comandos de [otros sistemas](#) por el hecho de que es simplemente un programa de usuario. y por el hecho de que se utiliza casi exclusivamente como mecanismo para invocar otros programas.

Las órdenes del sistema UNIX - de ahora en más, llamadas *comandos* - constan de una lista de cadenas o *palabras* interpretadas como el *nombre de comando* seguido de *argumentos*. De esta forma, el comando

```
mail bill
```

consta de dos palabras. La primera palabra `mai l` llama al comando que a ejecutar (en este caso el programa de correo que envía mensajes a otros usuarios). La shell recurre al nombre del comando

para cargarlo en memoria e iniciar su respectiva ejecución por usted. En este caso, buscará en varios *directorios* un archivo con el nombre `mail`, el cual usted espera que contenga el programa de correo electrónico.

Las demás palabras suministradas junto al comando se interpretan como sus *argumentos*, propicios para su ejecución específica. En este caso hemos también provisto el argumento `bill`, el cual el programa de correo electrónico `mail` interpreta como un nombre de usuario al cual se remitirá una pieza de correspondencia. Atendiendo a la impresión regular del terminal, el empleo del comando `mail` podría aparecer de la siguiente manera:

```
% mail bill
Tengo una pregunta en lo que refiere a la documentación de csh.
A mi documento parecería faltarle la sección 5.
Existe dicha sección?
EOT
%
```

Aquí se mecanografió un mensaje para remitírselo a Bill, concluyéndoselo mediante un `^D` (lo que transmite un mensaje de *fin de fichero* al programa de correo `mail`).

De ahora en más... la notación “`^x`” debe leerse “**Ctrl+x**” y representa presionar la **tecla x** mientras se mantiene presionada la **tecla Ctrl**.

A consecuencia, el programa de correo hizo eco de los caracteres “” y remitió nuestro mensaje. Los caracteres `%` antes y después del comando de correo fueron impresos por la C shell para indicar una solicitud de entrada.

Tras presentar el mensaje `%`, la shell leerá la entrada de órdenes desde nuestra terminal. Escribimos un comando completo `mail bill`. Luego, la shell ejecutó el programa de correo `mail` con el argumento `bill` y permaneció inactiva a la espera de su cumplimiento. A continuación, el programa de correo leyó la entrada de nuestra terminal hasta que le indicamos *fin del fichero* mecanografiando un `^D`, tras lo cual la shell notó que el correo se había completado y nos indicó que se encontraba presta para leer nuevamente desde la terminal, presentando inmediatamente otro `%`.

El patrón esencial de toda interacción con UNIX a través del shell es éste ciclo: se escribe una orden completa completa en el terminal, la shell ejecuta el comando y cuando su ejecución fue cumplida, solicita un nuevo comando. Si usas el editor durante una hora, la shell esperará pacientemente que termine de editar y aguardará obedientemente nueva orden suya una vez que cumplida la edición.

Un comando útil que puede ejecutar ahora como ejemplo es el comando `tset`, encargado de establecer los caracteres predeterminados para *borrar* y *eliminar* en su terminal: el carácter de *borrar* borra el último carácter que escribió y el carácter de *eliminar* elimina toda la línea que ha ingresado hasta el momento. Por defecto, el carácter borrar es `#` y el carácter eliminar es `@`. La mayoría de las personas que usan pantallas CRT prefieren usar el carácter de retroceso (`^H`) como carácter de borrado, ya que permite mayor facilidad para ver lo que ha escrito hasta ahora. Puede llevar a cabo este accionar mediante

```
tset -e
```

que ordena al programa *tset* configurar el caracter borrar, y su opción por defecto para este caracter es un caracter retroceder.

Argumentos de Opción

Una noción útil en UNIX es la del argumento de opción (“*flag*”). Si bien muchos argumentos de comandos especifican a nombres de archivo o nombres de usuario, existen ciertos argumentos capaces de especificar una capacidad opcional del comando (la cual desearías invocar en lugar de las opciones regulares). Por convención, tales argumentos *flag* comienzan con el caracter guion medio -. Por tanto, el comando

```
ls
```

producirá un listado de ficheros encontrados en el *directorio de trabajo* actual. El *flag* -s define a la opción de tamaño, por tanto

```
ls -s
```

ordena que *ls* también informe el tamaño (en bloques de 512 caracteres) de cada uno de los ficheros del listado. El Manual de Referencia de UNIX proporciona las opciones disponible para cada comando en su respectiva Sección. El comando *ls* - por ejemplo - consta de una gran cantidad de opciones útiles e interesantes. La mayoría de los demás comandos carecen de *flags*, o sólo disponen de una o dos. Como no resulta sencillo recordar las opciones de aquellos comandos que no se usan a menudo, la mayoría de las utilidades de UNIX suelen preferir realizar sólo una o dos funciones, evitando contar con muchas opciones difíciles de memorizar.

Salida a Ficheros

Los comandos que normalmente leen su entrada o escriben se salida en el terminal, también pueden ejecutarse con esta entrada y/o salida suplida a/desde un fichero.

Supongamos que deseamos guardar la fecha actual en un archivo llamado “ahora”.

La orden *date* imprime la fecha actual en el terminal. Esto se debe a que la *salida estándar* predeterminada para el comando *date* es nuestro terminal, por lo tanto, eso es lo que cumple. El shell nos permite *redirigir* la salida estándar de un comando por medio de una notación, si se recurre al *metacarácter* > y se invoca el nombre de fichero donde se colocará el resultado de salida. Por tanto, el comando

```
date > ahora
```

ejecuta el comando de *date* de modo que su *salida estándar* se dirija al fichero *ahora* en lugar de hacerlo al terminal. Por lo tanto, el resultado del comando cumple en colocar la fecha y hora actuales en el fichero *ahora*. Es importante saber que el comando *date* no sabe que su salida irá a un fichero en lugar de hacerlo al terminal: la shell realiza la *redirección antes* que el comando comience a ejecutarse.

Otra aspecto a tener en cuenta aquí es innecesario generar el fichero *ahora* previamente a la ejecución del comando *date*; si el fichero no existe, la shell lo crea. ¿Y si el archivo existiera? En tal caso, ¡sus contenidos previos habrían sido desechados! Existe una opción de shell llamada *noclobber* para evitar que esto accidentalmente suceda; se analiza en la [sección 2.2](#).

Normalmente, el sistema guarda los ficheros creados por usted con `>` junto a todos los demás ficheros. Por lo tanto, el resultado predeterminado es que los ficheros sean permanentes. Si desea crear un fichero para su eliminación automática, puede comenzar su nombre de fichero con un carácter `#`. Dicho carácter “numeral” denota el hecho de que el fichero será un *fichero borrador*. El sistema eliminará dichos archivos tras un par de días (o mas pronto si el espacio resulta limitado). Por lo tanto, al ejecutar el comando *date* como se indicó anteriormente, no queríamos que almacene la salida para siempre realmente, de manera que más probablemente usaríamos:

```
date > #ahora
```

Metacaracteres en la Shell

Como el caso del `>` indicado anteriormente, la shell tiene una gran variedad de [caracteres especiales](#) que ordenan funcionalidades especiales. Decimos que para la shell, dichas notaciones guardan significado *sintáctico* y *semántico*. Por lo general, para la shell la mayoría de los caracteres que no son letras o dígitos guardan un significado especial.

Los metacaracteres normalmente surten efecto sólo cuando la shell lee nuestra entrada [N.d.T. el teclado del terminal]. Como consecuencia, no debemos preocuparnos al usar metacaracteres en la redacción de una pieza que enviaremos a través de *mail*, o al mecanografiar textos o introducimos datos en cualquier otro programa. Debemos tener presente que la shell sólo lee entrada *cuando ha señalado su prompt* `%` .

Mas adelante habremos de aprender brevemente una manera de *citar* que nos permite recurrir a *metacaracteres* sin que la shell los trate de forma especial.

Entrada desde archivos; cañerías

Ya hemos aprendido como *redirigir* la *salida estándar* de un comando *a* un fichero. Es posible también redirigir la *entrada estándar* de un comando *desde* un fichero. Esto a menudo es innecesario pues la mayoría de los comandos leen desde un fichero siempre que su nombre sea provisto como argumento. Podemos dar el comando

```
sort < datos
```

para ordenar ejecución del programa *sort* redirigiendo su *entrada estándar* - el lugar desde donde el comando leerá normalmente su entrada – desde el fichero *datos*. Normalmente, recurriríamos a:

```
sort datos
```

Dejando que el comando *sort* abra un fichero *datos* por sí mismo para proveerse la entrada que necesita (ya que esto es más fácil de mecanografiar). Debemos notar que si sólo mecanografiamos:

```
sort
```

entonces el programa *sort* ordenaría las líneas desde la *entrada estándar*. Como en tal caso no redirigimos la *entrada estándar*, el reordenamiento de líneas se producirá tras que mecanografiar las líneas consecutivamente en el terminal, al presionar **^D** para indicar *Fin de Fichero*.

Una funcionalidad más útil es la posibilidad combinatoria que da redirigir la *salida estándar* de un comando con la entrada estándar de otro, por ejemplo para ejecutar los comandos en una secuencia conocida como *cañería*. Por ejemplo, el comando

```
ls -s
```

produce normalmente un listado de los ficheros de nuestro directorio, especificando el tamaño de cada uno (en bloques de 512 caracteres). Si nos interesa aprender cuál de nuestros ficheros es el más grande, podríamos querer ordenar este listado según su tamaño, en lugar de hacerlo alfabéticamente según su nombre (la cual es la manera en la que ordena el listado el programa *ls* por defecto).

Podemos observar las muchas opciones de *ls* para ver si consta de alguna opción que haga esto, pero eventualmente descubriríamos que no existe ninguna [N.d.T., en la versión de 1983 de *ls*; las versiones actuales la tienen, por supuesto]. En lugar de ello, podremos usar un par de *flags* simples del comando *sort* combinándolas con *ls* para lograr lo que nosotros queremos.

La opción *-n* de *sort*, especifica un ordenamiento numérico en lugar de un ordenamiento alfabético. Por tanto

```
ls -s | sort -n
```

especifica que la salida del comando *ls* argumentado con la opción *-S* deba ser *entubado* al comando *sort* argumentado según su opción de ordenamiento numérico. Esto nos proporcionará una lista de nuestros ficheros ordenada por tamaño (pero listando primero el más pequeño). Luego podremos utilizar la opción *flag* de ordenamiento inverso *-r* de *sort*, junto al comando *head* en combinación con aquél, ingresando:

```
ls -s | sort -n -r | head -5
```

Hemos tomado aquí un listado de nuestros ficheros ordenados alfabéticamente (cada uno con su tamaño en bloques), hemos pasado esto a la entrada estándar del comando *sort* solicitando que las ordene de forma numérica invertida (la más grande primero). Dicha salida ha sido vertida luego al comando *head*, encargado de presentar sólo las primeras pocas líneas (en este caso, las primeras 5). Como resultado, este comando cumple en presentar los nombres y tamaño de nuestros 5 ficheros más grandes.

La notación introducida anteriormente se conoce como mecanismo de *caño*. Los comandos separados por caracteres *|* resultan conectados entre sí por la shell y la *salida estándar* de cada uno de ellos es pasado a la *entrada estándar* del siguiente. El comando más a la izquierda en la cañería generalmente recibe su entrada estándar del teclado del terminal, y el comando más a la derecha volcará su salida estándar en el presentador del terminal. Mas adelante ofreceremos ejemplos de cañerías adicionales cuando discutamos el mecanismo historial; un uso importante de cañería allí ilustrado es el de redirigir información al impresor de líneas.

Nombres de Ficheros

Para ser ejecutados, muchos comandos necesitan estar argumentados con *nombres de fichero*. Los *nombres de ruta* de UNIX consisten en una cantidad de *componentes* separados por un caracter de barra */*. Con excepción del último componente, cada uno de ellos nombran un *directorio* en el cual reside el siguiente componente, denotando - en efecto - la *ruta* de directorios que debe seguirse para alcanzar el fichero en cuestión. Por lo tanto, la ruta

```
/etc/motd
```

especifica un fichero en el directorio `etc` que es un subdirectorio del directorio *raíz* `/`. Dentro de este directorio se encuentra el fichero denominado `motd` (significa “mensaje del día”). Se dice que un *nombre de ruta* comenzado con una barra `/` es *absoluto* pues que cuenta con la definición jerárquica completa desde su *raíz*. En tanto, los *nombres de ruta* que no comienzan con `/` se interpretan como parte del *directorio de trabajo* actual - por defecto se trata de su directorio *home* aunque puede cambiarlo dinámicamente con el comando de cambio de directorio `cd`. Se dicen que estos *nombres de ruta* son *relativos* al directorio de trabajo, ya que se descubren iniciando una búsqueda desde el directorio de trabajo, descendiendo a los niveles inferiores de la jerarquía de directorio siguiendo cada *componente* del nombre de ruta. Si el nombre de ruta carece de `/` alguna, entonces el fichero está contenido en el directorio de trabajo (y la ruta simplemente consiste en el *nombre de fichero* en el directorio de trabajo). Los nombres de ruta *absolutos* no tienen relación al directorio de trabajo.

La mayoría de los *nombre de ficheros* constan de una cantidad de caracteres alfanuméricos y puntos `.`. De hecho, es posible usar todos los caracteres impresos exceptuando la barra `/` en los nombres de archivo. *Es inconveniente* emplear la mayoría de los caracteres no alfabéticos en los nombres de fichero pues muchos de ellos guardan significado especial para la shell. El caracter punto `.` no es un metacaracter de la shell, y se utiliza a menudo para separar la *extensión* del nombre de archivo de la base del nombre. Por tanto

```
prog.c prog.o prog.errs prog.out
```

son cuatro ficheros relacionados. Comparten la porción *base* (una porción base es la parte del nombre que queda por delante del `.` final, y que sigue a los caracteres que no sean `.`). El fichero `prog.c` podría ser el código fuente de un programa de lenguaje C, el fichero `prog.o` podría el fichero de código objeto correspondiente, el fichero `prog.errs` los errores resultantes de una compilación del programa, y el fichero `prog.output` la salida de un programa compilado.

Si deseamos referirnos a los cuatro ficheros a la vez en un comando, podríamos utilizar la notación `prog.*`

Esta palabra *antes* que el comando del cual es un argumento sea ejecutado, resulta formulaicamente *expandida* por la shell a una lista de nombres que comienzan con `prog`, El caracter `*` aquí coincide con “cualquier secuencia de caracteres en un nombre de fichero” (incluyendo un secuencia vacía). Los nombres que coinciden resultan ordenados alfabéticamente y puestos en la *lista de argumentos* del comando. Por lo tanto el comando

```
echo prog.*
```

dará eco a los nombres

```
prog.c prog.errs prog.o prog.out
```

Note que los nombres están ordenados aquí, y en un orden distinto al que listamos arriba. El comando *echo* recibe cuatro palabras como argumentos, incluso aunque sólo mecanografiemos directamente una palabra como argumento. Las cuatro palabras fueron generados por la *expansión de nombre de fichero* a partir de una única palabra de entrada.

También hay disponibles otras notaciones para *expansión de nombre de fichero*. El caracter `?` coincide con “cualquier caracter único en un nombre de fichero”. Por tanto

echo `? ?? ???`

Dará como eco una línea de nombres de fichero; primero aquellos con nombres de un solo caracter, luego aquellos con nombres de dos caracteres, y finalmente aquellos con nombres de tres caracteres. Los nombres de cada longitud de caracteres reciben un orden independiente.

Otro mecanismo consiste en *entrecorchetado*, esto es, una secuencia de caracteres entre `[ y ]`. Esta metasecuencia coincide con “cualquier caracter único del conjunto entrecorchetado”. Por tanto

prog.`[co]`

coincidirá con

prog.c prog.o

del ejemplo anterior. También podemos poner dos caracteres rodeando un `-` en una notación para “caracteres definidos por rango”. Por tanto

cap.`[1-5]`

podría coincidir los ficheros

cap.1 cap.2 cap.3 cap.4 cap.5

si existiesen. Esto es la abreviación para

cap.`[12345]`

y es, por ello, equivalente en todo sentido.

Otro punto importante a notar es que si una lista de palabras argumento de comando (una *lista argumental*) contiene sintaxis de expansión de nombre de fichero, y si esta sintaxis de expansión de nombre de fichero falla en hacer coincidir cualquier nombre de fichero existente, en tal caso la shell considerará esto como error y presentará un diagnóstico “Sin coincidencia”:

No match.

y no cumplirá la ejecución del comando.

Otro punto muy importante es que los ficheros con el caracter de barra invertida `\` al comienzo del nombre son tratados especialmente. Ni el `*` ni el `?` o el mecanismo de entrecorchetado `[ y ]` encontrarán coincidencia. Esto impide coincidir accidentalmente los nombres `.` y `..` del directorio de trabajo - los cuales tienen significados especiales para el sistema - así como otros ficheros tales como [.cshrc](#), que no son visibles normalmente. El rol especial del fichero `.cshrc` será discutido más adelante.

Otro mecanismo de expansión de nombre de fichero ofrece acceso abreviado al nombre de ruta del directorio *home* de otros usuarios. Esta notación consiste en el caracter tilde `~`, seguida por el nombre de login de otro usuario. Por ejemplo, la palabra `~bill` dirigirá al nombre de ruta `/usr/bill` (siempre que el directorio *home* del tal *Bill* sea `/usr/bill`, claro está [N.d.T. actualmente, `/home/bill`]). Como en grandes sistemas, los usuarios pueden contar con varios

directorios de login distribuidos a lo largo de muchos volúmenes de disco diferentes, con diferente nombres de directorio prefijo, esta notación otorga una forma confiable de acceder a los ficheros de otros usuarios.

Un caso especial que incumbe a esta notación es el de un tilde ~ aislado, por ejemplo ~/mbox. Esta notación resulta expandida por la shell para representar al fichero mbox en su propio directorio *home*, por ejemplo, en /usr/bill/mbox para mi en Ernie Co-vax, la máquina [VAX](#) del Dpto. de Ciencias del Cómputo de la universidad de California Berkeley, donde fue preparado este documento. Esto puede resultar muy útil si usted ha usado *cd* para cambiar a otro directorio, y ha encontrado un fichero que desea copiar con *cp*. Si doy el comando

```
cp esefichero ~
```

la shell expandirá dicho comando a

```
cp esefichero /usr/bill
```

ya que mi directorio *home* es /usr/bill.

Existe también un mecanismo que usa los caracteres de abrir llave { y cerrar llave } para abreviar un “conjunto de palabras que constan de partes comunes” pero no pueden ser abreviadas a través de los mecanismos anteriormente nombrados ya que no son ficheros, o son nombres de ficheros que aún no existen y por lo tanto no pueden ser descriptos convenientemente aún. Tal mecanismo será descripto bastante mas adelante, en la [sección 4.2](#), pues se recurre a ellas con poca frecuencia.

Citado

Ya hemos visto cierta cantidad de metacaracteres respetados por la shell. Dichos metacaracteres tienen una complicación dificultosa: *no podemos usarlos directamente como partes de palabras*. Por lo tanto el comando

```
echo *
```

no dará como eco el caracter *. O bien dará como eco una lista ordenada de nombres de ficheros del *directorio de trabajo* actual, o bien imprimirá el mensaje **NO match** si en el directorio de trabajo no existen ficheros.

El mecanismo recomendado para poner aquellos caracteres que no son números, dígitos /, ., o - en una palabra argumento de un comando es apostrofarlo (encerrarlo entre apóstrofes ' . . . '). Por ejemplo:

```
echo '*'
```

Existe un caracter especial signo cierre de admiración ! que es es utilizado por el mecanismo *historial* de la shell que *no puede* ser anulado apostrofando de la forma anterior: '!'. Para *anular* su significado especial de la shell, tanto el signo de cierre de admiración ! como el caracter apóstrofo ' en sí *deben ser precedidos por un único caracter de barra invertida *. Por tanto

```
echo '\!'
```

```
imprime
```

```
'!'
```

Estos dos mecanismos resultan suficientes para poner cualquier caracter impreso en una palabra que sin que sean considerados un argumento de un comando de la shell. Pueden combinarse, como en

```
echo \ ' '*'
```

lo que imprime

```
'*'
```

ya que la primer barra invertida \ ha anulado el primer apóstrofo ' , mientras que el asterisco * aparecía entre apóstrofes ' .

Suspensión de Comandos

Existen varias maneras de forzar su interrupción tras ejecutar un comando y mientras la shell aguarda su cumplimiento. Por ejemplo, si ingresa el comando

```
cat /etc/passwd
```

El sistema presentará en su terminal una copia de un listado de todos los usuarios del sistema. Es probable que esta presentación continúe por varios minutos, a no ser que la interrumpa [N.d.T en la VAX 11/750 de Berkeley no era particularmente veloz y tenía miles de usuarios!]. Puede enviar una señal INTERRUPT al comando *cat* presionando la tecla DEL o RUBOUT de su terminal. La señal INTERRUPT provocará su finalización, pues *cat* no ha sido implementado para considerar dicha señal de otra manera diferente. La shell nota que el comando *cat* ha finalizado y ofrecerá nuevamente un prompt % . Si presiona INTERRUPT nuevamente, la shell simplemente repetirá su prompt en lugar de interrumpirse (lo que tendría el efecto de cerrar su sesión, al igual que *cat*). La shell interpreta las señales INTERRUPT y escoge continuar ejecutando comandos en lugar de finalizar.

Otra manera en la cual muchos programas pueden finalizar su ejecución es recibiendo un mensaje de *fin-de-fichero* a través de su entrada estándar. Este es el motivo por el cual el programa *mail* del primer ejemplo anterior terminaba cuando presionábamos ^D, lo que genera un *fin-de-fichero* desde la entrada estándar [N.d.T: el teclado]. La shell también finaliza cuando obtiene un *fin-de-fichero*, a lo que presentará el mensaje *logout*; UNIX entonces cierra su sesión de terminal. Esto significa que si ingresa demasiadas ^D, podría accidentalmente desligarse del sistema, la shell cuenta con un mecanismo para impedirlo. Esta opción *ignoreeof* será discutida en la [sección 2.2](#)

Si la entrada estándar de un comando se ha redirigido desde un fichero, entonces por lo general finalizará cuando el fichero que funciona de entrada llegue al final. Por tanto, si ejecutamos

```
mail bill < preparado.txt
```

el comando *mail* finalizará sin que ingresemos ^D. Esto se debe a que leerá el fichero *preparado.txt* hasta alcanzar su *fin-de-fichero*, el cual pusimos un mensaje para Bill con un programa editor. También podríamos haber hecho

```
cat preparado.txt | mail bill
```

pues el comando *cat* habría ofrecido el texto a través del entubado a la entrada estándar del comando *mail*. Una vez cumplido en presentar el fichero, el comando *cat* habría finalizado -

clausurando la tubería - y el comando *mail* recibiría un *fin-de-fichero* desde él, finalizando su operación. Si bien el resultado es el mismo, esta forma de utilizar un caño parece más complicado que redirigir un fichero a la entrada, de modo que es más probable usar la primer forma. Estos comando podrían también haber sido detenidos enviándoles un INTERRUPT.

Otra posibilidad para suspender un comando es suspender su ejecución de forma temporal, lo que nos posibilita continuar su ejecución posteriormente. Esto se realiza enviando una señal STOP por medio de ^Z. Esta señal provoca que la suspensión de todos los comandos en ejecución en el terminal - usualmente uno, pero podrían ser varios en caso de haberse recurrido a la ejecución de una tubería. En otro aspecto, no resultan afectados de manera alguna por la señal STOP.

Una vez que el comando original permanece suspendido, puede recurrirse a ejecutar otros comandos cualquiera. El comando suspendido puede ser continuado por medio del comando *fg* sin argumentos. La shell informa entonces cuál es el comando a continuar, y proseguirá con su ejecución. La suspensión no afecta de manera alguna la ejecución del comando, a no ser que cualquier fichero de entrada en uso del comando suspendido sea alterado de alguna manera durante la suspensión. Esta funcionalidad puede resultar muy útil durante la edición, de necesitarse revisar otro fichero antes de continuar. A continuación, se da un ejemplo de una suspensión de comando:

```
% mail haroldo
Alguien acaba de copiar un fichero enorme a mi directorio de usuario.
El fichero se denomina
^Z
Stopped
% ls
tremenda_broma
prog.c
prog.o
% jobs
[1] + Stopped
mail haroldo
% fg
mail haroldo
"tremenda_broma". ¿Sabes quién lo hizo?
EOT
%
```

En este ejemplo, un usuario está redactando una pieza de correo a haroldo pero olvida el nombre de un fichero que quería mencionarle. Suspende el comando *mail* ingresando ^Z. Una vez que la shell nota la suspensión del programa de correo, presenta **Stopped** y le ofrece un prompt, aguardando un nuevo comando. Acto seguido el individuo ingresa el comando *ls* para recordar el nombre del fichero en cuestión. Usa el comando *jobs* para descubrir cuál comando se encuentra suspendido, y consecuentemente ingresa el comando *fg* para continuar la ejecución del único programa detenido, el *mail* a haroldo. Luego continua dando entrada a *mail*, y una vez concluida la redacción, la finalizó con ^D, que denota el final del mensaje, ante lo cual el programa *mail* presenta de *fin-de-transmisión*.

El comando *jobs* presenta un listado de los comandos se encuentran suspendidos. La orden ^Z *solo debería mecanografiarse al comienzo de línea*, pues de enviarse dicha señal desde el teclado en cualquier otro lugar diferente al comienzo de línea, la shell descarta ("borra") toda la línea. Esto mismo sucede con las señales INTERRUPT y QUIT. Se ofrece mayor información sobre suspender trabajos y controlarlos se ofrece en la [sección 2.6](#).

Puede ser necesario recurrir a una interrupción de manera poco ortodoxa si se escribe o ejecuta programas que no están completamente depurados. Tal medida puede lograrse mecanografiando `^\\`, lo cual envía la señal QUIT. No resultaría extraño que esto provoque que la shell presente un mensaje similar a :

```
Quit (Core dumped)
```

denotando la creación de un fichero `core` conteniendo información sobre el estado del programa `a.out`, hecho sucedido no bien la señal QUIT produjo la terminación. Podrá examinar este fichero por usted mismo, o bien enviar información al autor del programa informándole dónde está el fichero `core`.

Si corre comandos en *segundo plano* (según lo explicado en la [sección 2.6](#)), entonces dichos comandos ignorarán las señales INTERRUPT y QUIT del terminal. Para detenerlas debe utilizar el comando *kill*.

Si desea examinar detenidamente la salida de un comando sin hacer que el mismo desaparezca de la pantalla (como cuando recurrimos a interrumpir la salida del comando `cat /etc/passwd`), podrá usar el comando

```
more | /etc/passwd
```

El programa paginador *more* pausa tras cada pantallazo, presentando el mensaje `- - More- -`, en cuyo momento usted podrá presionar la **barra espaciadora** para obtener otro pantallazo, o **Retorno** para obtener otra línea, o una **q** para finalizar el programa *more*. También puede usar *more* como filtro. Por ejemplo

```
cat /etc/passwd | more
```

funciona de forma similar al comando *more* de ejecución directa indicado anteriormente.

Para detener la presentación en el terminal de los comandos sin involucrar a *more*, podrá recurrir a las teclas `^S` para pausar la salida. Podrá continuar la presentación en el terminal mediante `^Q` (o cualquier otra tecla, si bien por lo general se utiliza `^Q` pues esta sólo reinicia la presentación de salida, sin convertirse en entrada de otros comandos posibles del programa que está usted ejecutando).

Podrá operar este método adecuadamente si usa terminales de baja velocidad, pero con 9600 baudios o más probablemente encontrará dificultoso ingresar `^S` y `^Q` lo suficientemente rápido como para pausar adecuadamente la salida. Este es el motivo por el cual por lo general se recurre al programa *more*.

Una posibilidad adicional es recurrir al caracter de *descartar-salida* `^O`. Al tipear este caracter, toda la salida del comando actual se descarta (rápidamente) hasta que se produzca la siguiente lectura de entrada, o bien hasta la aparición del siguiente prompt. Puede utilizar *descartar-salida* para permitir la finalización de un comando sin tener que cumplir la impresión de grandes cantidades texto de salida en un terminal lento. `^O` opera como un conmutador, de manera que es posible conmutar dicho descarte ingresando nuevamente `^O` mientras se produce el descarte de salida.

¿Ahora qué?

Hasta ahora hemos considerado varios mecanismos de la shell y aprendimos bastante sobre la manera en la cual esta funciona. Las secciones restantes avanzarán más en el funcionamiento interno de la shell, pero seguramente querrá probar usando la shell Csh antes de continuar.

La siguiente sección introducirá muchas funcionalidades particulares a Csh, de modo que debe cambiarse la shell a csh antes de comenzar a leerla.

Ahora está listo para intentar usar csh.

2. Detalles de la Shell para Usuarios de Terminal

Inicio de la shell y terminación

Al iniciar sesión en el sistema, éste comenzará la ejecución de una shell. Esta ingresará a su directorio *home* y comenzará por leer los comandos de un fichero llamado `.cshrc` localizado en dicho directorio. Todas las C shells que eventualmente inicie anidadas durante su sesión de terminal leerán este fichero también. Mas adelante veremos qué tipos de comandos es útil colocar allí, por ahora no es necesario contar con este fichero, y la C shell no se quejará por su ausencia.

La *shell de login* - aquella que se ejecuta tras su inicio de sesión al sistema leerá los comandos contenidos en el fichero `.cshrc` ya mencionado - tras lo cual leerá comandos contenidos en un fichero `.login` (también en su directorio *home*). Este fichero contiene comandos que usted desea ejecutar cada vez que inicie una sesión del sistema UNIX. Mi fichero `.login` contiene lo siguiente:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
    'set noglob ; eval `tset -s -m dialup:vt100 -m plugboard:?hp2621nl *`;
ts; stty intr ^C kill ^U crt
set time=15 history=10
msgs -f
if (-e $mail) then
    echo "Tiene correo! ${prompt}"
    mail
endif
```

Este fichero contiene varios comandos para que UNIX los ejecuta toda vez que inicie mi sesión de terminal. El primero de ellos es un *set*. Este comando es interpretado en forma directa por la C Shell, y configura la variable de shell `ignoreeof`, que impide que la C shell desligue sesión al presionar **^D**. En lugar de ello utilizo el comando *logout* para cerrar la sesión del sistema. Al establecer la variable `mail`, le pido a la C shell que revise los correos entrantes por mí; cada 5 minutos la C shell revisa este fichero y me informa si me ha llegado correo electrónico. Una alternativa en lugar de usar tal *set* es colocar en su lugar el comando

`biff y`

lo que hará que se me notifique inmediatamente al arribo de correo (mostrándome unas pocas primeras líneas del nuevo correo).

Posteriormente configuré la variable de shell `time` a `15`, lo que hace que la shell imprima automáticamente líneas de estadística para aquellos comandos cuya ejecución requiera al menos 15 segundos de tiempo de CPU. La variable `history` se establece en `10`, indicando que deseo que la C shell recuerde los últimos 10 comandos mecanografiados en su *listado de historial* (descrito mas adelante).

Creé un *alias* “`ts`” que ejecuta un comando `tset` estableciendo los modos de terminal. Los parámetros de `tset` indican los tipos de terminal que uso a menudo cuando no estoy en el puerto normal de la VAX. Luego ejecuto `ts` y también uso el comando `stty` para cambiar el caracter de interrupción a `^C` y el caracter de eliminar línea a `^U`.

Luego corro el programa `msgs` que me provee cualquier mensaje de sistema que no hubiese visto antes; la opción `-r` le impide decirme cualquier cosa si no hay mensajes nuevos. Finalmente, de existir mi fichero de casilla de correo, entonces ejecuto el programa `mail` para procesar mi correspondencia.

Cuando los programas `mail` y `msgs` terminan, la shell finalizará de procesar mi fichero `.login` y comenzará a leer comandos desde el terminal, saludando a cada uno de ellos con el prompt `%`. Cuando cierro sesión (ejecutando el comando `logout`) la C shell imprimirá `logout` y ejecutará comandos del fichero `.logout` si éste está presente en mi directorio *home*. Cumplido aquello, la shell terminará y UNIX me desconectará del sistema. Si el sistema no se apaga, recibiré un nuevo mensaje de inicio de sesión. En cualquier caso, luego del mensaje `logout` la shell estará conminada a terminar, y no tomará en cuenta otra entrada desde mi terminal.

Variables de la Shell

La shell mantiene un conjunto de *variables*. Vimos anteriormente las variables `history` y `time` que tenían valores de `10` y `15`. De hecho, cada variable de shell cuenta como valor un arreglo de cero o mas *cadena*s. Las variables de la shell pueden recibir valores por medio del comando `set`. Tiene varias formas, la más útil de las cuales fue indicada anteriormente y es

```
set nombre=valor
```

Las variables de shell pueden utilizarse para almacenar valores que deben ser usados por posteriores comandos a través del mecanismo de sustitución. Las variables de shell más comúnmente referenciadas son aquellas a las cuales la shell misma refiere. Sin embargo, al modificar los valores de dichas variables, uno puede afectar directamente el comportamiento de la shell.

Una de las variables más importantes es la variable `path`. Ella contiene una secuencia de nombres de directorio que la shell analizará al buscar comandos. El comando `set` sin argumentos presenta los valores asignados de todas las variables definidas (usualmente decimos “establecidas”) en la shell. Los valores por defecto de `path` presentados por `set` son

```
% set
argv      ( )
cwd       /usr/bill
home      /usr/bill
path      ( . /usr/ucb /bin lusr/bin )
prompt    %
shell     /bin/csh
status    0
term      vt100
```

```
user      bill
%
```

Esta salida indica que la variable `path` apunta al directorio actual `.`, y luego a los directorios `/user/ucb`, `/bin` y `/usr/bin`. Los comandos que pueden escribir podrían estar en `.` (usualmente uno de sus directorios). Los comandos desarrollados en Berkeley residen en `/usr/ucb`, mientras que los comandos desarrollados en los [Laboratorios Bell](#) residen en `/bin` y `/usr/bin`.

Cierta cantidad de programas desarrollados localmente en el sistema residen en el directorio `/usr/local`. Si deseamos que todas las shells que invocamos tengan acceso a estos nuevos programas, podremos establecerlos en nuestro fichero `.cshrc` en nuestro directorio *home* mediante el comando

```
set path=(. /usr/ucb /bin /usr/bin /usr/local)
```

Intente hacer esto y tras cerrar sesión y volver a iniciarla, ejecute `set` nuevamente para comprobar que el valor establecido a `path` ha cambiado.

Una cosa que debería tener presente es que la shell examina cada directorio que ha establecido en su `path` y determina cuáles comandos están allí contenidos. Exceptuando el directorio actual `.` - el cual la shell trata especialmente - esto significa que si se agregan comandos a un directorio en su `path` *posteriormente* a la iniciar la shell, no serán necesariamente encontrados por la shell. De desear utilizar un comando que agregado de esta manera, deberá indicar primero el comando

```
rehash
```

para refrescar que la shell, recalculando su tabla interna de locaciones de comandos, que se utiliza en la búsqueda de los comandos agregados recientemente. Ya que toda vez que ejecuta un comando, la shell pesquisa en el directorio actual `.`, establecerlo en la `path` al principio de la especificación opera por lo general de forma equivalente, reduciendo los tiempos de espera.

Otras variables incorporadas útiles consisten en las variables `home`, que muestra su directorio *home*, `cwd` que contiene su directorio de trabajo actual, y la variable `ignoreeof` (que dice a la shell que no se desligue al recibir un carácter *fin de fichero* desde un terminal (como se mencionó anteriormente, lo cual puede establecerse en su fichero `.login`). La variable `ignoreeof` es una de aquellas en las que la shell hace caso omiso de su valor asignado, sólo analiza si dice `set` o `unset` denotando si está establecida o no, respectivamente. Por lo tanto, para establecer esta variable se indica simplemente

```
set ignoreeof
```

y para desestablecerla, se indica

```
unset ignoreeof
```

Los comandos anteriores no otorgan valor alguno a la variable `ignoreeof`, pues ésta no los requiere ni los desea en lo absoluto.

Finalmente, algunas de las otras variables incorporadas de la shell en uso son las variables `noclobber` y `mail`.

La metasintaxis

> fichero

que redirige la salida estándar de un comando, *sobrescribirá y destruirá* los contenidos previos de **fichero**, y como tal es posible sobrescribir accidentalmente un fichero valioso. Si prefiere que la shell no sobrescriba ficheros de esta manera tan liviana, puede indicar en su fichero **.login**

set noclobber

Tras loguearse nuevamente, intente ordenar

date > ahora

y comprobará un *diagnóstico de error* si **ahora** ya existe. Para producir la sobrescritura de **ahora** si *realmente* lo desea, debe mecanografiar

date >! ahora

El >! es una metasintaxis especial que indica que está bien “darle una paliza” a **fichero** [NdT: clobber significa “apalear”, y es la jerga utilizada para denotar la sobrescritura inadvertida del **fichero**].

El espacio en blanco entre el ! y la palabra **ahora** es crítico, ya que **!ahora** invocaría las [mecanismos de historial](#), que tiene un efecto completamente distinto.

Historial de la shell

La C Shell puede mantener una *lista de historial* en la cual coloca las palabras de los comandos ya ejecutados. Es posible reutilizar comandos o palabras de los comandos en la formación de nuevos comandos a través de una notación. Tal mecanismo puede utilizarse para repetir comandos previos o corregir errores de mecanografiado menores en los comandos.

A continuación se ofrece una sesión de ejemplo que involucra el uso típico del mecanismo de historial de la C Shell. En tal ejemplo se recurre a un terminal para revisar con *cat* el contenido del fichero **bug.c**, un programa muy simple en lenguaje C que contiene uno o dos bugs. Luego intentamos [correr el compilador CC](#) con él, refiriendo el fichero nuevamente a través de la orden **!\$** (lo que significa “el último argumento del comando anterior”).

Aquí, **!** es el metacaracter de invocación al mecanismo historial, mientras que **\$** refiere al “argumento final”, en analogía al uso de **\$** en el editor *ed*, que significa “final de la línea”).

La C Shell da eco del comando tal como se hubiese mecanografiado sin recurrir al mecanismo de historial, y luego lo ejecuta. Ya que la compilación produce un *diagnóstico de error*, se ejecuta el editor en el fichero que se intentó compilar, se corrige el bug, y se ordena repetir la ejecución del compilador de C con el mecanismo de historial. Para ello, esta vez se simplemente indicamos el comando **!c**, que significa “repetir el último comando historiado que comenzaba con la letra **c**”). De haber existido otros comandos historiados que comenzaban con la letra **c**, se podría haber tenido que recurrir a especificar más detalladamente, ya sea con **!cc**, o incluso **!cc:p** (que hubiese ordenado “imprime el último comando que comenzaba con **cc**, sin ejecutarlo”).

```

% cat bug.c
main ()
{
printf("hola");
}

% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error

% ed !$
ed bug.c
29
4s/);/"&/p
printf("hola");
w
30
q

% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/la/la\\n/p
printf("hola\n");
w
32
q

% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b

% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill      3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill      3932 Dec 19 09:42 bug

% bug
hola

% num bug.c | spp
spp: Command not found.

% ^spp^ssp
num bug.c | ssp
1 main()
3 {
4
printf("hola\n");
5 }

% !! | lpr
num bug.c | ssp | lpr
%
```

Tras recompilar, se ejecutó el fichero `a.out` resultante, y se notó que un bug aún persistía, por lo cual se corrió nuevamente el editor. Tras corregir el programa se corrió el compilador C

nuevamente, asociado esta vez al comando extra `-o bug`, que indica al compilador colocar el binario resultante en un fichero `bug` (en lugar de utilizar el nombre `a.out` al que recurre el compilador por defecto).

El mecanismo de historial puede usarse por lo general en cualquier lugar de la formación de nuevos comandos y pueden colocarse otros caracteres antes y después de los comandos sustituidos.

Luego se ejecutó el comando `size` para comprobar cuán grande eran las imágenes binarias resultantes de los programas compilados, y luego se recurrió al comando `ls -l` con la misma *lista argumental*, denotando la lista argumental `..`. Finalmente, se ejecutó el programa `bug` para comprobar si su salida es de hecho correcta.

Se ejecutó el comando `num` con el fichero `bug.c` para obtener un listado numerado del programa. Acto seguido se quiso entubar la salida de `num` a través del filtro `ssp` a fin de comprimir las líneas en blanco, pero se cometió un error de mecanografía, escribiendo erróneamente `spp`. Para corregirlo el yerro se recurrió a un mecanismo de sustitución de shell, colocando la cadena errónea y la correcta entre caracteres `^` (pues opera de igual forma al comando “sustituir” del editor `ed`). Finalmente, se reiteró el mismo comando con `!!`, pero enviando su salida a la impresora de línea.

Existen otros mecanismos para reiterar comandos. El comando `history` imprime cierta cantidad de comandos previos asociados a una numeración, por medio de la cual se los puede referenciar.

Hay una manera de referir a un comando previo buscándolo a por medio de una cadena que aparezca en el, y existe otra - menos útil - para seleccionar argumentos a incluir en un nuevo comando. Se ofrece en las páginas [man](#) de `csh` una descripción completa de todos estos mecanismos, así como en el Manual de Programador de UNIX.

Alias

La C shell cuenta con un mecanismo de *alias* que puede ser utilizado para efectuar transformaciones en los comandos de entrada. Puede utilizarse este mecanismo para simplificar los comandos que interesan, supliendo a dichos comandos los argumentos deseados por defecto, o bien para operar transformaciones a los comandos y sus argumentos.

La funcionalidad de alias es similar a una facilidad de macros.

Algunas de las funcionalidades obtenidas por medio de alias también pueden lograrse usando ficheros de intérprete de comandos, pero estas se llevarán a cabo en otra instancia anidada de la C shell, y no podrán afectar directamente el ambiente de la shell actual o involucrar comandos tales como `cd` (que deben correr en la shell actual).

Por ejemplo, supongamos que existe una nueva versión del programa de correo en el sistema que se llama *newmail*, la cual deseamos utilizar en lugar del programa de correo estándar *mail*. Si en su fichero `.cshrc` dispone el siguiente comando de shell

```
alias mail newmail
```

la C shell transformará una línea de entrada dada según la forma

```
mail bill
```

en una llamada de ejecución a `newmail`.

En concreto, supongamos ahora que deseamos el comando `ls` presente siempre los tamaños de los ficheros en su listado (o sea, que `ls` utilice siempre el argumento `-S`). Podemos usar

```
alias ls ls -s
```

o incluso

```
alias dir ls -s
```

creando una nueva sintaxis de comando llamada `dir` que ejecute un comando `ls -s`. Si decimos `dir ~bill`

entonces la C shell lo traducirá como

```
ls -s /mnt/bill
```

De esta forma, el mecanismo de *alias* puede usarse para proveer nombres cortos a comandos, para proveer argumentos por defecto, y para definir nuevos comandos cortos en relación a otros comandos. También es posible definir alias que contengan múltiples comandos o cañerías, mostrando donde se encuentran los argumentos del comando original para sustituirlos por medio de las facilidades del mecanismo de historial. Por tanto la definición

```
alias cd 'cd \!* ; ls '
```

haría un comando `ls` tras efectuar todo cambio de directorio con el comando `cd`. Apostrofamos entre caracteres `'` la definición entera del alias a fin de impedir que se produzcan la mayoría de las substituciones, y que el caracter `:` sea reconocido como un metacaracter. En este caso, al introducir el comando de alias, este `!` resulta anulado con una `\` para impedir que resulte interpretado. Aquí, el `\!*` sustituye la *lista argumental* completa del comando `cd` previo al alias (sin dar un error si no tuviese argumentos). El `;` que separa los comandos aquí se utiliza para indicar que se debe cumplir un comando, y luego el siguiente. De forma similar, la definición

```
alias whois 'grep \!^ /etc/passwd'
```

define un comando que busca su primer argumento en el fichero de contraseñas.

Cuidado: La shell lee el fichero `.cshrc` cada vez que inicia. Si dispone allí una cantidad excesiva de comandos, las shells tenderán a tener un inicio lento. Se encuentra en desarrollo un mecanismo útil capaz de guardar el ambiente de la shell luego de leer el fichero `.cshrc` y restaurarlo rápidamente, pero por ahora debe intentar limitar la cantidad de alias establecidos a un número razonable... 10 o 15 es razonable; 50 o 60 serán casuales de notables retrasos de inicialización de las shells, con lo que el sistema se volverá lento al ejecutar comandos anidados desde el interior del editor u otros programas.

Mas redirecciones: `>>` y `>&`

Aún existen varias notaciones adicionales útiles para el usuario de terminal que no fueron explicadas: las redirecciones adicionales.

Además de la salida estándar ya tratada, los comandos cuentan también con una *salida de diagnóstico* que normalmente está dirigida al terminal - incluso si la salida estándar está redirigida a un fichero o a un caño. En ocasiones es deseable dirigir la salida de diagnóstico junto con la salida estándar. Por ejemplo, si desea redirigir la salida de un comando de larga ejecución a un fichero, y desea contar con un registro de cualquier diagnóstico de error que produzca (“log”), puede ingresar comando `>& fichero`

Aquí el `>&` le solicita a la C shell que redirija tanto la *salida de diagnóstico* como la *salida estándar* a `fichero`.

De manera similar, puede dar la orden comando

comando `|& lpr`

para redirigir tanto la salida estándar y la salida de diagnóstico a través de un caño al demonio de impresión `lpr`.

Finalmente, es posible usar la forma

comando `>> fichero`

para colocar la salida al final de un `fichero` existente.

Noclobber

Existe una forma de comando

comando `>&! fichero`

Esta se usa cuando la [variable noclobber](#) está activada y `fichero` ya existe. Si `noclobber` se encuentra activada, da como resultado un error `fichero no existe`. De otro modo la shell creará `fichero` si este no existe. La forma

comando `>>! fichero`

hará no dará error si `fichero` es inexistente y la variable de shell `noclobber` se encuentra establecida (activada).

Trabajos; Segundo Plano, Primer Plano, o Suspendido

Al introducir uno o más comandos juntos siguiendo la forma de *cañería* (separados por `|`) o *secuencia de comandos* (separados por punto y coma `;`), la shell genera un único *trabajo* (“job”), que consiste en dichos comandos unificados. Un comando aislado (sin `|` o `;` genera) un trabajo simple. Por lo general, cada línea mecanografiada a la shell genera un trabajo. Algunas de las órdenes que generaron trabajos (uno por línea) fueron

```
sort < datos
ls -s | sort -n | head -5
mail haroldo
```

Toda vez que al final de la orden se mecanografía el metacaracter `&`, entonces la misma se iniciará en la forma de *trabajo en segundo plano*. Esto significa que la shell no aguardará su cumplimiento, sino que quedará inmediatamente a la espera de otro comando. Dicho trabajo se ejecutará *en*

segundo plano. Esto guarda diferencia con aquellos trabajos normales - llamados *trabajos en primer plano* - que siguen siendo leídos y ejecutados por la shell de uno a la vez. Por lo tanto

```
du > uso_de_disco &
```

corre el programa *du*, encargado de reportar el uso de disco de su directorio de trabajo (así como el de cualquier directorio por debajo de él), coloca su salida en el fichero *uso_de_disco* y vuelve inmediatamente la atención a usted indicando un prompt para eventual disposición de un ulterior comando nuevo (sin aguardar la finalización de *du*). Mientras tanto el programa *du* continuará su ejecución en segundo plano hasta finalizar, incluso aunque usted mecanografie y solicite cumplimiento de otros comandos. Al finalizar un trabajo en segundo plano, la shell presentará un mensaje informando tal cosa justo antes de su prompt siguiente.

En el siguiente ejemplo, el trabajo *du* finalizó en cierto momento durante la ejecución del comando *mail*, y su finalización se reportó justo antes de que el trabajo *mail* finalizase.

```
% du > uso_de_disco &
[1] 503

% mail bill
¿Cómo es posible saber cuando un trabajo en segundo plano ha finalizado?
EOT
[1] - Done          du > uso_de_disco

%
```

Ante una finalización anormal de un trabajo, el mensaje **DONE** podría intercambiarse por otro diferente, tal como **Killed**. Si usted desea que se le reporten aquellas finalizaciones de programas en segundo plano al mismo momento en que se producen - eventualmente incluso interrumpiendo la salida de terminal de otros trabajos que pudiese tener en primer plano - puede establecer la variable *notify*. Siguiendo el ejemplo anterior, podría implicar que el mensaje **DONE** eventualmente aparezca directamente en medio de la redacción del correo a Bill.

Los trabajos en segundo plano no resultan afectados por ninguna señal del teclado tales como las señales **STOP**, **INTERRUPT**, o **QUIT** mencionadas anteriormente.

La C shell inicia el alta los trabajos en una *tabla de trabajos* incorporada, y finaliza su baja con su cumplimiento. En esta tabla de trabajos la C shell tabula el nombre de comando, argumentos, *número de identificador proceso* (PID) de todos los comandos, así como el directorio de trabajo donde se inició cada trabajo. Cada trabajo tabulado puede estar

- En ejecución *en primer plano* (la C shell aguarda su finalización),
- En ejecución *en segundo plano*,
- *suspendido*.

Sólo puede estar en ejecución en primer plano un trabajo por vez, pero es posible contar con varios trabajos suspendidos o corriendo en segundo plano a la vez. Los números de trabajo siguen siendo los mismos hasta que el trabajo finalice, y luego son reutilizados.

Al ordenar ejecución de un trabajo en segundo plano por medio del sufijo *&*, antes que la shell le presente un prompt para ingresar un nuevo comando le informará el *número de cola de trabajos* - así como los números de identificador de proceso de todos sus comandos de nivel superior - . Por ejemplo,

```
% ls -s | sort -n > uso_de_disco &
[2] 2034 2035
%
```

ejecuta el programa *ls* con la opción -S, entuba su salida al programa *sort* con la opción -n, y coloca la salida resultante en el fichero *uso_de_disco*. Debido al sufijo & al final de la línea ambos programas se iniciaron conjuntamente en la forma de un *trabajo en segundo plano*. Una vez iniciado el trabajo, la shell presentó el *número de cola de trabajos* entre corchetes - [2] en este caso - seguido por el *número de proceso* de cada programa iniciado en el trabajo en cuestión. Inmediatamente después, la shell ofrece un prompt para ingresar eventualmente una nueva orden, dejando al trabajo en ejecución concurrente (en simultáneo).

Como se mencionó en la [sección 1.8](#), los trabajos en *primer plano* se suspenden mecanografiando ^Z, lo cual envía una señal STOP al trabajo en ejecución en primer plano. Puede suspender un trabajo en ejecución en segundo plano mediante comando *stop* que se describirá mas adelante. Una vez que los trabajos se suspenden, su ejecución se detiene hasta ser iniciados nuevamente - ya sea en el primer o en el segundo plano. La shell toma nota de la suspensión de un trabajo, e informa el hecho de manera bastante similar a los informes de finalización de trabajo en segundo plano. En el caso del trabajo en primer plano, esto guarda una apariencia similar a

```
% du > uso_de_disco
^Z
Stopped
%
```

en donde la shell presenta el mensaje **Stopped** al notar que el programa *du* se ha suspendido. Al usar el comando *stop* para detener los trabajos en segundo plano, se presenta

```
% sort uso_de_disco &
[1] 2345

% stop %1
[1] + Stopped (signal)      sort uso_de_disco

%
```

De necesitar alternar temporalmente entre lo que estamos haciendo, puede ser útil suspender los trabajos en primer plano, ejecutar otros comandos, y retornar al trabajo suspendido. Asimismo, es posible suspender los trabajos en primer plano y continuar su ejecución en la forma de trabajos en segundo plano por medio del comando *bg*. Esto permite *continuar un trabajo distinto*, poniendo el segundo plano en espera hasta que el trabajo en primer plano finalice. Por tanto

```
% du > uso_de_disco
^Z
Stopped

% bg
[1] + uso_de_disco &

%
```

inicia *du* en el primer plano, lo detiene antes de finalizar, usa *bg* para pasarlo a ejecución en segundo plano, permitiendo así eventuales nuevos comandos de ejecución en primer plano. Esto se presenta especialmente útil cuando un trabajo en primer plano demanda un tiempo de ejecución

mayor que el previsto, por lo que usted hubiese deseado iniciarlo en segundo plano desde un principio.

Comandos de control de trabajos

Comando de Control de Trabajos	Resultado
jobs	Presenta la cola de trabajos
fg	Trae un trabajo en ejecución en segundo plano o suspendido a ejecución en primer plano
bg	Pasa un trabajo suspendido a segundo plano
stop, Ctrl+Z	Suspende un trabajo en ejecución en segundo plano
kill	termina un trabajo suspendido o en segundo plano o
notify	notifica inmediatamente cuando finalizan los comandos

Todos los *comandos de control de trabajos* de la C shell pueden recibir un argumento que identifica el trabajo particular. Los *argumentos de nombre de trabajo* deben indicarse con el caracter %, si bien alguno de los comandos de control del trabajo también aceptan números de identificador de proceso PID (presentados por el comando *ps*). El trabajo por defecto (si no se le proveen argumentos) es llamado *trabajo actual* y es identificado por un + in la salida del comando *jobs*, el cual muestra cuáles son los trabajos que tiene. Cuando sólo existe un trabajo en ejecución o detenido en el segundo plano (el caso usual), el mismo siempre será el *trabajo actual*, y por tanto no necesita argumentos. Si un trabajo ha sido detenido en el primer plano, se transforma en el *trabajo actual* y el trabajo actual existente se convierte en el trabajo *previo*. Cuando es dado, el argumento es tanto %- (que denota *trabajo previo*; %# donde el # es el número de trabajo; %pref (donde pref es algún prefijo único del nombre de comando y argumentos de uno de los trabajos; o ?% seguido por alguna cadena encontrada en solo uno de los trabajos, presenta la tabla de trabajos con los tipos de comandos de trabajos, los comandos, el número de trabajo, y estado de status (Stopped o Running) de cada trabajo en segundo plano o trabajo suspendido. Sumando la opción -l también presenta los números de identificador de proceso PID.

```
% du > uso_de_disco &
[1] 3398
```

```
% ls -s | sort -n > mifichero &
[2] 3405
```

```
% mail bill
^Z
Stopped
```

```
% jobs
[1] - Running      du > uso_de_disco
[2]  Running      ls -s | sort -n > mifichero
[3] + Stopped     mail bill
```

```
% fg
```

```
% ls
ls -s | sort -n > mifichero

% more mifichero
```

El comando *fg* pasa un trabajo suspendido o en segundo plano a ejecución en primer plano. Se utiliza para reiniciar un trabajo previamente suspendido, o cambiar un trabajo suspendido para que corra en el primer plano (permitiendo el arribo de señales o entrada desde el terminal). En el caso anterior, se utilizó *fg* para pasar el trabajo *ls* desde el segundo plano al primer plano pues se deseaba aguardar su finalización antes de estudiar su fichero de salida. El comando *bg* reinicia ejecución de un trabajo suspendido en el segundo plano. Usualmente se utiliza tras suspender con la señal STOP el actual trabajo en ejecución en primer plano (combinar la señal STOP con el comando *bg* conmuta un trabajo de ejecución en primer plano a un trabajo en ejecución en segundo plano). El comando *stop* suspende un trabajo en segundo plano.

El comando *kill %nro_de_co* la extermina inmediatamente un trabajo en segundo plano o trabajo suspendido. Además de los trabajos, puede recibir como argumento números de procesos, tal como son presentados por *ps*. Por tanto, en el ejemplo anterior, al correr el comando *du* podría haber sido terminado por el comando

```
% kill %1
[1] Terminated      du > uso_de_disco
%
```

El comando *notify* (no es la variable *notify* mencionada anteriormente) indica que debe informar inmediatamente el cumplimiento y finalización de un trabajo específico inmediatamente (en lugar de aguardar el siguiente prompt).

Si un trabajo en ejecución en segundo plano intenta leer entrada del terminal, resulta detenido automáticamente. Posteriormente, al pasar dicho trabajo a primer plano, podrá recibir entrada. Si se lo desea, es posible enviar nuevamente el trabajo al segundo plano hasta que éste solicite nuevamente entrada. Dicho procedimiento se ilustra en la siguiente secuencia de sesión, donde el comando *S* del editor de texto puede requerir un tiempo para cumplir.

```
% ed ficherogigante
120000
1,$s/estapalabra/otrapalabra
^Z
Stopped

% bg
[1] ed ficherogigante &

%
... algunos comandos en segundo plano
[1] Stopped (tty input)      ed ficherogigante

% fg
ed ficherogigante
w
120000
q
%
```

Por tanto, tras indicar el comando `S`, el trabajo *ed* fue detenido con `^Z` y luego pasado al segundo plano usando *bg*. Cierta tiempo después el comando `S` finalizó, *ed* intentó leer otro comando y resultó detenido pues los trabajos en segundo plano no pueden leer desde el terminal. El comando *fg* devolvió al primer plano el trabajo *ed*, donde una vez más pudo aceptar comandos desde el terminal.

El comando

```
stty tostop
```

detiene todos los trabajos en segundo plano que intenten escribir salida en el terminal. De esta manera se impide que los mensajes de los trabajos en segundo plano interrumpen la salida de los trabajos en primer plano, y permiten correr un trabajo en segundo plano sin perder la salida de terminal. También puede utilizarse con programas interactivos que presentan a menudo períodos largos sin interacción. Por tanto, se detendrá cada vez que una salida solicite mayor entrada del terminal, antes de presentar el prompt de solicitud. Podrá entonces correrlo en primer plano por medio *fg* a fin de proveer tal entrada, y si es necesario detenerlo y pasarlo al segundo plano. Si no desea tener salida de trabajos en segundo plano que interrumpen su trabajo, podrá encontrar útil insertar este comando *stty* en su fichero `.login`. También puede reducir la necesidad de redirigir la salida de los trabajos en segundo plano, y si la salida no es muy grande:

```
% stty tostop

% wc ficherogigante &
[1] 10387

% ed texto
... algún tiempo después
q
[1] Stopped (tty output)    wc ficherogigante

% fg wc
wc ficherogigante
13371 30123 302577

% stty - tostop
```

En el ejemplo anterior, se ejecutó el comando `wc` (que resulta en una salida de sólo una línea que cuenta líneas, palabras y caracteres de un fichero). Como este requeriría aguardar un cierto tiempo, se detuvo su salida antes que intentara imprimir en el terminal, y nos pusimos a editar el fichero `texto` para aprovechar el tiempo. Luego, se ha permitido que `wc` - con su trabajo cumplido - imprima su línea en el terminal, exactamente en el momento en que estábamos listos para observar la salida. Aquellos programas que intentan cambiar el modo del terminal también se bloquearán toda vez que no se encuentren en primer plano (ya sea que *tostop* esté establecido o no, ya que sería muy inconveniente que un programa en segundo plano quiera cambiar el estado del terminal).

Como los comandos de trabajos sólo presentan aquellos trabajos *iniciados en la shell de ejecución actual*, desconocen de los trabajos en segundo plano iniciados en otras sesiones de login, o aquellos trabajos ocurridos dentro de los ficheros de la shell. En este caso puede usarse *ps* para descubrir los trabajos en segundo plano no iniciados durante la instancia actual de shell.

Directorios de Trabajo

Como se mencionó en [sección 1.6](#), la shell siempre se encuentra activa en un *directorio de trabajo* particular. El comando de “cambio de directorio” *chdir* (también puede usarse su abreviatura *cd*) modifica el directorio de trabajo de la shell - esto es, cambia el directorio en el cual usted se encuentra actualmente.

Es útil crear un directorio para cada proyecto en el que desea trabajar, y disponer en él todos los ficheros relacionados a dicho proyecto. El comando “crear directorio” *mkdir* genera un directorio nuevo. El comando *pwd* (“imprimir directorio de trabajo”) informa la ruta absoluta de directorio del directorio de trabajo de la shell - esto es, el directorio en el cual se encuentra actualmente. Por lo tanto, en el siguiente ejemplo

```
%pwd
/usr/bill

% mkdir periodico

% chdir periodico

%pwd
/usr/bill/periodico

%
```

el usuario ha creado y se ha posicionado en el directorio *periodico*, donde - por ejemplo - podría colocar un grupo de ficheros relacionados. Siempre podrá volver a su directorio *home* de inicio de sesión - sin importar donde se ha movido en la jerarquía de directorios - simplemente ingresando

```
cd
```

sin mas argumentos. El nombre *..* siempre significa “el directorio por encima de la jerarquía del directorio actual”, por lo tanto

```
cd ..
```

Cambia el directorio de trabajo de la shell al directorio por encima del actual. El nombre *.* puede ser usado en cualquier *nombre de ruta*. Por lo tanto

```
cd ../programas
```

significa cambiar al directorio *programas* contenido en el directorio por encima del directorio actual. Si tiene varios directorios para diferentes proyectos - por debajo, por tomar de ejemplo, de su directorio *home* - esta notación abreviada le permitirá cambiar rápidamente entre ellos.

La shell siempre recuerda el nombre de ruta de directorio de trabajo actual a través de su variable *cwd*. La shell también puede ser requerida de recordar el directorio previo, al cambiar a un nuevo directorio de trabajo. Si en lugar del comando *cd* se utiliza el comando “empujar directorio” *pushd*, la shell guardará el nombre del directorio de trabajo actual en un listado de *pila directorios de uso previo* al cambio de directorio actual. Podrá consultar dicho listado en cualquier momento ingresando el comando “directorios”, *dirs*.

```
% pushd periodico/referencias
-/periodico/referencias -
```

```
% pushd /usr/lib/tmac
/usr/lib/tmac ~/periodico/referencias ~

% dirs
/usr/lib/tmac ~/periodico/referencias ~

% popd
~/periodicos/referencias ~

%.popd
%
```

El listado de la pila de directorios de uso previo resultará impreso en una línea horizontal, que se lee de izquierda a derecha, con una abreviatura de su directorio `home` (en este caso `/usr/bill`) en forma de tilde (`~`). La pila de directorio se imprime toda vez que hay mas de una entrada en ella, y se ve alterada. También es impresa por un comando *dirs*. Generalmente *dirs* resulta más rápido e informativo que *pwd* ya que junto al directorio de trabajo actual, muestra como cualquier otro directorio registrado en la pila de directorios de uso previo.

El comando *pushd* sin argumentos alterna el directorio actual, con el primer directorio en la pila de directorios de uso previo. El comando “traer directorio” *popd* sin argumento lo devuelve al directorio donde se encontraba con anterioridad al actual, descartando el directorio de trabajo previo de la pila de directorios. Si ingresa *popd* varias veces en serie, retrocederá entre los directorios que ha estado navegando (caminando hacia atrás) por el comando *pushd*. Existen otras opciones a *pushd* y *popd* para manipular los contenidos de la pila de directorios de uso previo, y cambiar a directorios que no se encuentran en la parte superior de la pila; para mas detalles consulte las manpages de Csh.

Como la shell recuerda el directorio de trabajo en el cual ha comenzado cada trabajo, le advertirá de manera práctica cuando podría confundirse al reiniciar un trabajo en el primer plano que tiene un directorio de trabajo diferentes que el directorio de trabajo actual de la shell. Por lo tanto, al iniciar un trabajo en segundo plano, esta cambiará entonces el directorio de trabajo de la shell y luego conmutará a ejecución en primer plano el trabajo que se encontraba en segundo plano. La shell advertirá entonces que el directorio de trabajo del trabajo en actual ejecución en primer plano es diferente al tabulado en la tabla de trabajos de la C shell.

```
% dirs -l
/mnt/bill

% cd miproyecto

% dirs
~/miproyecto

% ed prog.c
1143
^Z
Stopped

% cd ..

% ls
miproyecto
fichero_texto

% fg
```

```
ed prog.c (wd: ~/miproyecto)
```

De esta forma la shell le advierte con **wd: directorio** en el momento que exista discrepancia por un cambio de directorio de trabajo, incluso aunque no se haya utilizado un comando *cd*. En el ejemplo anterior, el trabajo suspendido **ed** aún se encontraba en */mnt/bill/proyectos*, incluso aunque la shell se había cambiado a *mnt/bill*. Una advertencia similar se ofrece cuando tal trabajo en primer plano finaliza o resulta suspendido (usando la señal STOP). Retornar nuevamente a la shell implica cambiar de directorio de trabajo nuevamente, indicado por **wd now: directorio**.

```
% fg
ed prog.c (wd: ~/miproyecto)
... tras alguna edición
q
(wd now: ~)

%
```

Estos mensajes resultan algo confusos si usted usa programas que cambian sus propios directorios de trabajo, pues C shell sólo recuerda desde cuál directorio un trabajo se ha iniciado, y asume su permanencia allí. La opción *-l* de *jobs* presentará el directorio de trabajo de los trabajos suspendidos o en segundo plano cuando discrepan del directorio de trabajo actual de la shell.

Presentar un prompt con el directorio actual de trabajo en la shell - como es usanza en los BSD modernos - solventa en gran medida estas problemáticas. En los 80s se evitaba pues el comando *cwd* requería un par de segundos en los lentos discos de la VAX, y enlentecía todo el sistema multiusuario [N.d.T.].

Comandos Incorporados útiles

Ahora ofrecemos algunos pocos comandos útiles incorporados en la shell, y describimos como se usan.

El comando *alias* descrito arriba se usa para asignar nuevos alias y para mostrar los alias existentes. Si argumentos, presenta los alias existentes. También puede dársele sólo un argumento tal como

```
alias ls
```

El comando *echo* imprime sus argumentos. A menudo es usado en guiones de shell o como un comando interactivo para ver qué expansiones de nombre de fichero producirá.

El comando *history* mostrará los contenidos de su *listado de historial*. Los números dados a los eventos del historial pueden usarse para referencia eventos anteriores que son difíciles de referenciar utilizando los mecanismos contextuales introducidos anteriormente. Existe también una variable de shell denominada **prompt**.

Al colocar un caracter **!** en su valor, la shell substituirá el número del comando actual con el del *listado de historial*. Puede usar este número para referir a este comando en una substitución de historia. Por lo tanto, podría

```
set prompt='\!% '
```

Tenga presente que el `!` debe ser *anulado* aquí incluso entre caracteres de apoóstrofo `'`.

El comando *limit* se use para restringir el uso de recursos de cómputo. Sin argumentos, presentará las limitaciones actuales:

```
cputime      unlimited
filesize     unlimited
datasize     5616 kbytes
stacksize    512 kbytes
coredumpsize unlimited
```

Los límites pueden establecerse, de esta manera:

```
limit coredumpsize 128k
```

Funcionarán la mayoría de las abreviaciones razonables; vea el man de *cs*h para más detalles.

El comando *logout* puede utilizarse para terminar una sesión de shell que tiene establecido la variable *ignoreeof*.

El comando *rehash* provoca que la shell recompute una tabla de localización de comandos. Esto es necesario si se agregó un comando a un directorio en la ruta de búsqueda *path* de la shell actual, y se desea que la shell proceda a encontrarla, de otro modo el algoritmo de detección podría decirle a la shell que el comando no se encontraba en los comandos indicados cuando la tabla de hash fue computada originalmente.

El comando *repeat* puede utilizarse para repetir un comando varias veces. Por tanto, para realizar cinco copias del fichero *uno* hasta el fichero *cinco*, podría hacer

```
repeat 5 cat uno >> cinco
```

El comando *setenv* puede usarse para establecer variables de ambiente. Por tanto

```
setenv TERM=adm3a
```

establecerá el valor de la variable de ambiente *TERM* a la cadena *adm3a*.

Un programa de usuario llamado *printenv* existe, que imprimirá el ambiente. Podría mostrar

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=./usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

El comando *source* puede usarse para forzar a la shell actual a leer comandos desde un fichero. Por tanto

```
source .cshrc
```

puede usarse luego de editar un cambio en el fichero *.cshrc* el cual desea que surta efecto antes de la siguiente vez que inicie sesión.

El comando *time* puede usarse para cronometrar un comando, sin emportar cuántos ciclos de computadora requiera. Por tanto

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
    52 178 1347 /etc/rc
    52 178 1347 /usr/bill/rc
   104 356 2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

indica que el comando *cp* usó una cantidad de tiempo de usuario negligible (u) y aproximadamente 1/10 del tiempo de sistema (s); el tiempo transcurrido fue de 1 segundo (0:01). Hubo un uso promedio de memoria de 2k bytes de espacio de programas, y 1k bytes de espacio de datos sobre el tiempo de CPU involucrado (2+1k), el programa hizo tres lecturas de disco y dos escrituras (3+2io), y tomó un paginado y no hizo uso de memoria de intercambio (1pf+0w). El comando de conteo de palabras *wc* por otro lado utilizó un décimo de segundos de tiempo de usuario (0.1), menos de un segundo del tiempo transcurrido en el sistema. El porcentaje de 13% indica que durante dicho período el comando activo *wc* usó un 13 por ciento de los ciclos de computadora disponibles en la máquina.

Los comandos *unalias* y *unset* pueden usarse para remover alias y definiciones de variable de la shell, y *unsetenv* quita variables del ambiente.

¿Qué mas?

Esto concluye la discusión básica de la shell para los usuarios de terminal. Existen más funcionalidades de la shell que las explicadas aquí, y todas las funcionalidades de la shell se tratan en las manpages. Una funcionalidad útil que se discutirá luego es el comando incorporado *foreach* que puede utilizarse para ejecutar la misma secuencia de comandos con una cantidad de argumentos distintos.

Si usted desea utilizar UNIX, debería revisar el resto de este documento y las manpages de la shell para familiarizarse con las otras funcionalidades que tiene disponibles.

3. Estructuras de Control de Shell y Guiones de Comandos

Introducción

Es posible colocar comandos en ficheros y hacer que la shell se involucre leyendo y ejecutando los comandos que contienen tales ficheros, a los cuales se les llamará [script](#) de shell. Detallamos aquí aquellas funcionalidades de la shell que resultan útiles a quienes escriben tales scripts.

Make

Es importante recalcar primero para qué son útiles los guiones de shell. Existe un programa llamado *make* que es muy útil para mantener un grupo de ficheros relacionados, desarrollando un conjunto de operaciones sobre ellos. Por ejemplo, un gran programa consistente en uno o mas ficheros puede tener sus dependencias descritas en un *makefile*, un fichero que guarda definiciones de comandos utilizados para crear esos ficheros diferentes cuando las definiciones cambian. Estas definiciones facilitan los medios para imprimir listados, limpiar el directorio en el cual residen los programas, instalar el programas resultante colocando mas apropiadamente los ficheros definidos en

este makefile. Tal formato es superior y preferible a manejar un grupo de procedimientos de shell afrotar dicha tarea.

De la misma manera, al realizar preparaciones documentales puede crearse un `makefile` para definir cómo crear las versiones diferentes del documento y cuáles opciones de `rooff` o `troff` son adecuadas para su utilización.

Invocación y variable `argv`

Puede interpretar un guion de C Shell diciendo

```
% csh guion ...
```

donde `guion` es el nombre del fichero que contiene un grupo de comandos de Csh y ... se reemplaza por una secuencia de argumentos. La shell coloca dichos argumentos en la variable `argv` y luego comienza a leer los comandos del guion. Estos parámetros están puestos a disponibilidad a través de los mismos mecanismos que se usan para referenciar cualquier otra variable de shell.

Si hace el fichero `guion` ejecutable haciendo

```
chmod 755 guion
```

y coloca un comentario de shell, al comienzo del guion de shell (ej, comienza el fichero con un caracter de #), seguido de `/bin/csh`, esta automáticamente será invocada para ejecutar el guion cuando ingrese

```
guion
```

Si el fichero no comienza con # entonces se utilizará la shell estándar `/bin/sh` para ejecutarlo. Esto le permitirá convertir sus guiones más antiguos para interpretarlos con csh según lo crea conveniente.

Sustitución de Variables

Una vez que cada línea de entrada es dividida en palabras y se realizan las substituciones de historia, se desmenuza la línea de entrada en comandos unívocos. Antes de ejecutar cada comando, en las palabras se lleva a cabo un mecanismo conocido como *sustitución de variables*. Indicado por el caracter \$, esta sustitución reemplaza los nombres de las variables por sus valores. Por tanto el

```
echo $argv
```

colocado en un guion de comandos, provocará que el valor de la variable `argv` sea presentada como salida del eco del guion de shell. Si `argv` no se encuentra establecida en este momento, provocará un error.

Están previstas varias notaciones de acceso a componentes y atributos de variables. La notación `$?nombre`

se expande a 1 si `nombre` está *establecida* (`set`), o en 0 si *no está establecida* (`unset`). Este es el mecanismo fundamental para evaluar si las variables particulares han sido asignadas a valores. Todas las demás formas de referencia a variables no definidas provocarán errores.

La notación

`$#nombre`

se expande al número de elementos en la variable `nombre`. Por tanto

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

También es posible acceder a componentes de una variable que cuenta de varios valores. POr tanto

`$argv[1]`

da el primer componente de `argv` o - en el ejemplo anterior - `a`. De igual forma

`$argv [$#argv]`

dará `c`, mientras que

`$argv[1-2]`

dará `a b`. Otras notaciones útiles en guiones de shell son

`$n`

siendo `n` un entero que oficia de abreviación de

`$argv[n]`

el `nº` parámetro, y

`'$*`

que as la abreviación de

`$argv`

La forma

`$$`

expande al número de proceso de la shell actual. Como dicho número de proceso es único en el sistema, bien puede utilizarse para generar nombres de fichero temporales únicos. La forma

`$<`

es bastante especial y es reemplazada por la siguiente línea de entrada, a ser leída desde la *entrada estándar* de la shell (no del guion que estaba siendo leído). Esto resulta muy útil para escribir

guiones de shell interactivos, que leen comandos de la terminal, o incluso escribir guiones de shell que actúan como un [filtro](#), capaz de leer líneas de su *fichero de entrada*. Por tanto, la secuencia

```
echo '¿si o no?\c'  
set r- ($<)
```

Presentará la solicitud `¿si o no?`, sin presentar una nueva línea, y luego leerá una respuesta dada, colocándole como variable `r`. En este caso `$#r` será `0` si se mecanografía tanto una línea en blanco o un *fin-de-fichero* (`^D`).

Debemos llamar la atención de una diferencia menor entre `$n` y `$argv[n]`. La forma `$argv[n]` dará un error si `n` no se encuentra en el rango `1- $#argv`, mientras que `$n` nunca dará un error de suscripto fuera de rango. Esto se hace para retener compatibilidad con la manera con la cual [los shells mas antiguos](#) manejaban los parámetros.

Otro punto importante es que nunca es un error dar un subrango en la forma de `n-`; si hay menos de `n` componentes de una variable dada entonces no se substituye palabra alguna. Un rango en la forma `m-n` de la misma forma devolverá un vector vacío sin dar error, toda vez que `m` exceda el número de elementos de la variable dada, provisto que el suscripto `n` se encuentre dentro del rango.

Expresiones

Para poder construir guiones de shell interesantes, debe ser posible evaluar expresiones en la shell basadas en los valores de variables. De hecho, todas las operaciones aritméticas del lenguaje C están disponibles en la shell con la misma precedencia que tienen en el lenguaje C. En particular, las operaciones `==` y `!=` comparan cadenas y los operadores `&&`, y `||` implementan las operaciones de lógica de Boole AND/OR. Los operadores especiales `' '` y `!'` son similares a `==` y `!=`, excepto que la cadena en el lado derecho puede tener caracteres de coincidencia de patrones como `*`, `?`, o `[ ]`, y la evaluación consistirá en si la cadena de la izquierda coincide con el patrón de la derecha.

La shell también permite solicitudes de ficheros según la forma

```
-? nombre_fichero
```

donde `?` es reemplazado por una cifra numérica de un caracteres simples. Por ejemplo, la expresión primitiva

```
-e nombre_fichero
```

informará si el fichero `nombre_fichero` existe. Otras primitivas evalúan por acceso de lectura, escritura y ejecución de un fichero., si este es un directorio, o si tiene longitud no cero.

Es posible evaluar si un comando termina normalmente, por una primitiva de la forma de `{ comando }`, que retorna *verdadero* (ej, `1`) si el comando tiene éxito en salir normalmente con `status 0` de salida, o `0` si el comando termina anormalmente o con un status de salida distinto a cero. De requerirse mayor información detallada sobre el status de ejecución de un comando, puede ser ejecutado con la variable `$status` examinada en el siguiente comando. Como `$status` está establecida por todos los comandos, es muy demostrativa. Puede ser salvada si es inconveniente de usarla sólo una sola vez inmediatamente por el comando que sigue.

Si desea un listado completo de componentes de expresión disponible, observe dicha sección del manual de la shell.

Ejemplo de guion de shell

A continuación, un guion de shell de ejemplo, que hace uso de mecanismo de expresión de la shell y alguna de sus estructuras de control

```
% cat copyc.sh
#
# Copyc.sh copia los programas en la lista especificada
# al directorio ~/respaldo si difieren de los ficheros
# que ya están en ~/respaldo
#
set noglob
foreach i ($argv)
    if ($i != *.c) continue # no es un fichero .c de modo que no hace nada
    if (! -r ~/respaldo/$i:t) then
        echo $i:t no en respaldo ... no cp\`ado
        continue
    endif

    cmp -s $i ~/respaldo/$i:t # para establecer $status

    if ($status != 0) then
        echo nuevo respaldo de $i
        cp $i ~/respaldo/$i:t
    endif
end
```

Este guion hace uso del comando *foreach*, que provoca que la C Shell ejecute los comandos dispuestos entre el *foreach* y su correspondiente *end* para cada uno de los valores dados entre la (y la) con la variable nombrada, en este caso *i* establecida en valores sucesivos de la lista. Dentro de este bucle podemos usar el comando *break* para detener la ejecución del bucle y continuar para terminar prematuramente una iteración y comenzar con la siguiente. Luego del bucle *foreach* la variable de iteración (*i* en este caso) tendrá el valor de la última iteración.

Establecemos la variable *noglob* aquí para impedir que la expansión de nombre de ficheros de los miembros de *argv*. Esto es generalmente una buena idea si los argumentos de un guion de shell son nombres de ficheros que han sido ya expandidos, o si los argumentos podrían contener metacaracteres de expansión de nombres de fichero. También es posible citar cada uso de una variable de expansión \$, pero esto es más difícil y menos confiable.

La otra estructura de control empleada aquí es una declaración de la forma

```
if ( expresión ) then
    comando
    ...
endif
```

La colocación de las palabras claves aquí utilizada no es flexible, debido a la actual implementación de la shell (ver [limitaciones](#)).

La shell tiene otra forma de la declaración *if* que guarda la forma

```
if ( expresión ) comando
```

La cual - si es una línea muy larga - puede escribirse

```
if ( expresion ) \  
    comando
```

Aquí anulamos la nueva línea por estética. El comando no debe involucrar un |, & o ;, y no debe ser otro comando de control. La segunda forma, requiere que la \ final preceda inmediatamente al *fin-de-línea* (o sea, *no de debe dejarse espacio*).

La declaración *if* más general anterior también admite la secuencia o pares *else-if* seguidos por un único *else* y un *endif*, por ejemplo

```
if ( expresion ) then  
    comandos  
else if (expresion ) then  
    comandos  
else  
    comandos  
endif
```

Otro mecanismo importante usado en guiones de shell es el modificador :. Podemos usar el modificador :r aquí para extraer una raíz de un nombre de fichero, o :e para extraer la extensión. Por tanto, si la variable i tiene el valor /mnt/foo.bar, entonces

```
% echo $i $i:r $i:e  
/mnt/foo.bar /mnt/foo bar  
%
```

muestra como el modificador :r quita el .bar final de la cadena, y el modificador :e deja solo el bar. Otros modificadores quitan el último componente de un *nombre de ruta*, dejando la cabeza :h o dejando todo a excepción del último componente del *nombre de ruta* dejando la cola :t. Estos modificadores son descriptos al completo en las manpages de csh, en el manual del programador. Es también posible usar el mecanismo de sustitución de comandos descriptos en la siguiente sección para desarrollar modificaciones en cadenas y luego reingresar el ambiente de la shell. Como cada uso de este mecanismo involucra la creación de un nuevo proceso, es mucho más costoso en términos computacionales que usar la modificación :. Finalmente, debemos notar que el caracter # introduce léxicamente un comentario de shell en los guiones de shell (por no desde la terminal). Todos los caracteres subsecuentes en la línea de entrada a posterior del # resultan descartados por la shell. Este caracter puede ser citado usando * o \ para ponerlo en una palabra argumento.

Limitaciones

Las siguiente dos formas no son aceptables por C shell:

```
if ( expresión )      # No funciona!  
then  
    comando  
    ...  
endif
```

y

```
if ( expresión ) then comando endif      # No funciona
```

También es importante notar que la implementación de la C shell limita el número de modificadores : en una sustitución \$ a solo 1. Por tanto

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

no hace lo que uno esperaría.

Otras estructuras de control

La C shell también cuenta con estructuras de control `while` y `switch` similares a las del lenguaje C. Estas guardan las formas

```
while ( expresión )
    comandos
end
```

y

```
switch ( palabra )
```

```
case str1:
    comandos
    breaksw
...
```

```
case stm:
    comandos
    breaksw
```

```
default:
    comandos
    breaksw
```

```
endsw
```

Para mayores detalles, consulte la sección del manual de `cs`. Los programadores de C deben notar que usamos `breaksw` para salir de un `switch`, mientras que `break` sale de un bucle `while` o `foreach`. Un error común de cometer en los guiones de `cs` es usar los conmutadores `break` en lugar de `breaksw`.

Finalmente, `cs` permite declaraciones `goto` con etiquetas que son similares a las de C. Ejemplo:

```
bucle:
    comandos
goto bucle
```

Suplir entrada a comandos

Los comandos ejecutados desde guiones de shell reciben por defecto la entrada estándar de la shell que corre el guion. Esto es distinto a shells precias que corren en UNIX. Esto permite a los guiones de C shell de participar completamente en tuberías, pero obligan a realizar notación extra para los comandos que deben tomar datos en línea.

Por lo tanto necesitamos una metanotación para proveer datos en línea a los comandos situados en guiones de C shell. Como ejemplo, considere este guion que corre el editor para borrar los caracteres iniciales en blanco desde las líneas en cada fichero argumento.

```
% cat deblanqueador.sh
# deblanqueador.sh - Quita los caracteres en blanco al inicio de una línea
foreach i ($argv)
ed - Si << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
%
```

La notación << 'EOF' significa que la *entrada estándar* para el comando *ed* debe provenir del texto en el fichero de guion de shell en sí, hasta la siguiente línea, lo que literalmente consiste en “'EOF'”. El hecho de que se encuentre apostrofado ([citado](#)) provoca que la shell no sustituya variables en las líneas intervinientes. En general, si alguna parte de la palabra que sigue a « - la cual la shell usa para terminar el texto a ser dado al comando - se cita, entonces se anula la substitución de tales substituciones. En este caso, como en nuestro guion de edición hemos utilizado la forma 1, \$, debemos asegurar que este \$ no sufra substitución como variable. Podríamos haberlo asegurado también por medio de preceder dicha \$ con un caracter de \. Por ejemplo:

```
1,\$s/^[ ]*//
```

sin embargo, citar el terminador 'EOF' es una manera más confiable de lograr lo mismo.

Atrapando interrupciones

Si nuestro guion de shell crea ficheros temporales, podríamos querer atrapar interrupciones del guion de shell de modo de poder limpiar dichos ficheros. Podremos entonces hacer

```
onintr label
```

donde *etiqueta* es una etiqueta en nuestro programa. Si la shell recibe una interrupción, hará un *goto etiqueta* y podremos remover los ficheros temporales y luego hacer un comando *exit* (que está incorporado en la shell) para salir del guion de shell. Si deseamos salir con *status no cero*, podremos hacer

```
exit(1)
```

por ejemplo, para salir con status 1.

¿Qué mas?

Existen otras funcionalidades de la shell útiles a quienes escriben procedimientos de shell. las opciones *verbose* y *echo* y sus respectivas opciones de línea de comandos relacionadas -l y -x, pueden usarse para asistir en al rastreo de acciones de shell. La opción -n hace que la shell sólo lea comandos *sin ejecutarlos*, y en ocasiones puede resultar de suma utilidad.

Es importante notar que csh no ejecuta guiones de shell que no comiencen con el caracter # - lo que significa que los guiones de C shell comienzan con un comentario. De forma similar, el /bin/sh

de su sistema podría bien diferir del `csH` en la interpretación de guiones de shell que comiencen con `#`.

Esto es lo que permite que tanto los guiones de shell y la C shell vivan en armonía.

También existe otro mecanismo de citado que usa `"`, que permiten que se ejecuten sólo algunos mecanismos de expansión que hemos discutido hasta el momento en la cadena citada, y sirve para hacer que esta cadena se convierta en una palabra única, como lo hace `'`.

4. Otras funcionalidades de la shell menos utilizadas

Bucles en la terminal; variables como vectores

En ocasiones es útil usar la estructura de control *foreach* en el terminal para asistir en la realización de iteraciones de comandos similares. Por ejemplo, en el sistema Cory UNIX en Cory Hall hubo en un momento tres shells en uso: `/bin/sh`, `/bin/nsh`, y `/bin/csh`. Para contar la cantidad de personas que usaban cada shell, uno podría haber enviado los comandos

```
% grep -c csh$ /etc/passwd
27

% grep -c nsh$ /etc/passwd
128

% grep -c -v sh$ /etc/passwd
430

%
```

Ya que estos comandos son muy similares, podremos usar un *foreach* para simplificar.

```
% foreach i ('sh$' 'csh$' -v 'sh$')
? grep -c $i /etc/passwd
? end
27
128
430

%
```

Tenga aquí presente que al leer desde el cuerpo de un bucle, la shell le solicita entrada con un `?`. Las variables son de utilidad en los bucles, pues puede asignarle listas de nombres de ficheros u otras palabras. Por ejemplo, podría hacer

```
% set a=('ls')

% echo $a
csh.n csh.rm

% ls
csh.n
csh.rm

% echo $#a
2

%
```

Aquí el comando `set` da a las variables de una lista `a` cuyo valor es todos los nombres de fichero del directorio actual. Podemos luego iterar sobre dichos nombres para realizar cualquier función que escojamos.

La salida de un comando apostrofado entre caracteres `'` es convertida por la shell a una lista de palabras. También puede entrecomillar la cadena citada entre caracteres `"` para considerar cada línea *no vacía* como un componente de variable (impidiendo que las líneas resulten divididas como palabras a lo largo de sus caracteres de espacio en blanco o tabuladores. Existe un modificador `-x` que puede ser utilizado luego para expandir cada componente de variable en otra variable, dividiéndola en palabras separadas a lo largo de sus caracteres de espacio en blanco o tabuladores.

Llaves {...} en expansión de argumentos

Otra forma de expansión de nombre de fichero aludida anteriormente involucra el uso de caracteres de llave abierta `{` y llave cerrada `}`. Dichos caracteres especifican que la cadena contenida entre llaves, separada por comas `,`, debe ser sustituida consecutivamente en los caracteres contenidos entre llaves, y los resultados, expandidos de izquierda a derecha. Por lo tanto

```
A{cad1,cad2... stm}B
```

se expande a

```
Acad1B Acad2B... AstmB
```

Esta expansión ocurre antes que otras expansiones de nombre de fichero, y puede aplicarse recursivamente (si se encuentra anidada). Los resultados de cada cadena expandida se ordenan por separado, preservándose el orden de izquierda a derecha. No se requiere que existan los nombres de fichero resultante si no se emplean mecanismos de expansión ulteriores. Esto significa que este mecanismo puede utilizarse para generar argumentos que no sean nombres de fichero, pero que tengan partes en común.

Un uso típico de esto sería

```
mkdir ~/ {hdrs,retrofit,csch}
```

que crea los subdirectorios `hdrs`, `retrofit` y `csch` en su directorio `home`. Este mecanismo es más útil cuando el prefijo común es más largo que el dado en este ejemplo, por ejemplo:

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

Sustitución de Comandos

Un comando encerrado en caracteres resulta reemplazado justo antes de que se expandan los nombres de fichero, por la salida de dicho comando. Por lo tanto, es posible hacer

```
set pwd='pwd'
```

para salvar el directorio actual en la variable `pwd`, o bien hacer

```
ex 'grep -l TRACE *.c'
```

para ejecutar el editor `ex` proveyéndolo de aquellos ficheros cuyos nombres finalicen en `.c` y que contengan la cadena `TRACE` como argumento.

La expansión de comandos también ocurre en la entrada redirigida con «y dentro de citado entrecomillado con """. Refiérase al manual de la shell para los detalles completos

Otros detalles no cubiertos

En circunstancias particulares puede ser necesario conocer la naturaleza y orden exacto de las diferentes sustituciones desarrolladas por la shell. El significado exacto de ciertas combinaciones de citas es importante ocasionalmente. Estas se detallan al completo en su respectiva sección del manual. La shell tiene una cantidad de opciones *flags* que se emplean mayoritariamente para escribir programas de UNIX, y depurar guiones de shell. Consulte la sección de manual de la shell para obtener un listado de dichas opciones.

Caracteres Especiales

La siguiente tabla lista los caracteres especiales de csh y el sistema UNIX.

Metacaracteres sintácticos

Metacaracter sintáctico	Sintáctica
;	separa comandos para ejecutarlos secuencialmente
	separa comandos en un caño
()	entre paréntesis de expresiones y valores de variables
&	continúa ejecutando los siguientes comandos sin esperar completar la ejecución

Metacaracteres de nombre de fichero

Metacaracter de nombre de fichero	Acción
/	separa los componentes de una ruta de fichero
?	expansión de caracter coincidente con cualquier caracter único
*	expansión de caracter coincidente con cualquier secuencia de caracteres
[]	expansión de secuencia coincidente con cualquier caracter único de un conjunto
~	usado al comienzo del nombre de fichero para indicar directorios \$home
{ }	usado para especificar grupos de argumentos con partes comunes

Metacaracteres de citado

Metacaracter de citado	Acción
/	impide el metasignificado del siguiente caracter único
'	impide el metasignificado de un grupo de caracteres
*	como ' pero permite variables y expansión de comandos

Metacaracteres de entrada y salida

Metacaracter de redirección	Redireccionado
<	indica redirección de entrada
>	indica redirección de salida

Expansión/substitución de metacaracteres

Metacaracter	Acción
\$	indica substitución de variables
!	indica substitución de historial
:	precede modificadores de substitución
^	usado en formas especiales de substitución de historial
`	indica substitución de comandos

Otros metacaracteres

Metacaracter	Resultado
\$	comienza garabateo de nombres de fichero; indica comentarios de shell
!	opción de refijado (flag) argumentos a comandos.
prefija especificaciones de nombre de trabajo	